Genuine atomic multicast on Apache Kafka

E. Steinmacher, D. Pasadakis, E. De Lima Batista, O. Schenk, F. Pedone, & P. Eugster

Universitá della Svizzera Italiana, Institute of Computing, Lugano, Switzerland.
Universitá della Svizzera Italiana, Computer Systems Institute, Lugano, Switzerland.

PASC24 Zurich Switzerland 3-5 June 2024

Scaling throughput within Apache Kafka

➤ **Apache Kafka** is an event streaming platform that guarantees total order (TO) within a partition but not across partitions.[1]
➜ **Proof of concept: Expand the Apache Kafka framework with TO multicast across partitions.**

Partitioning

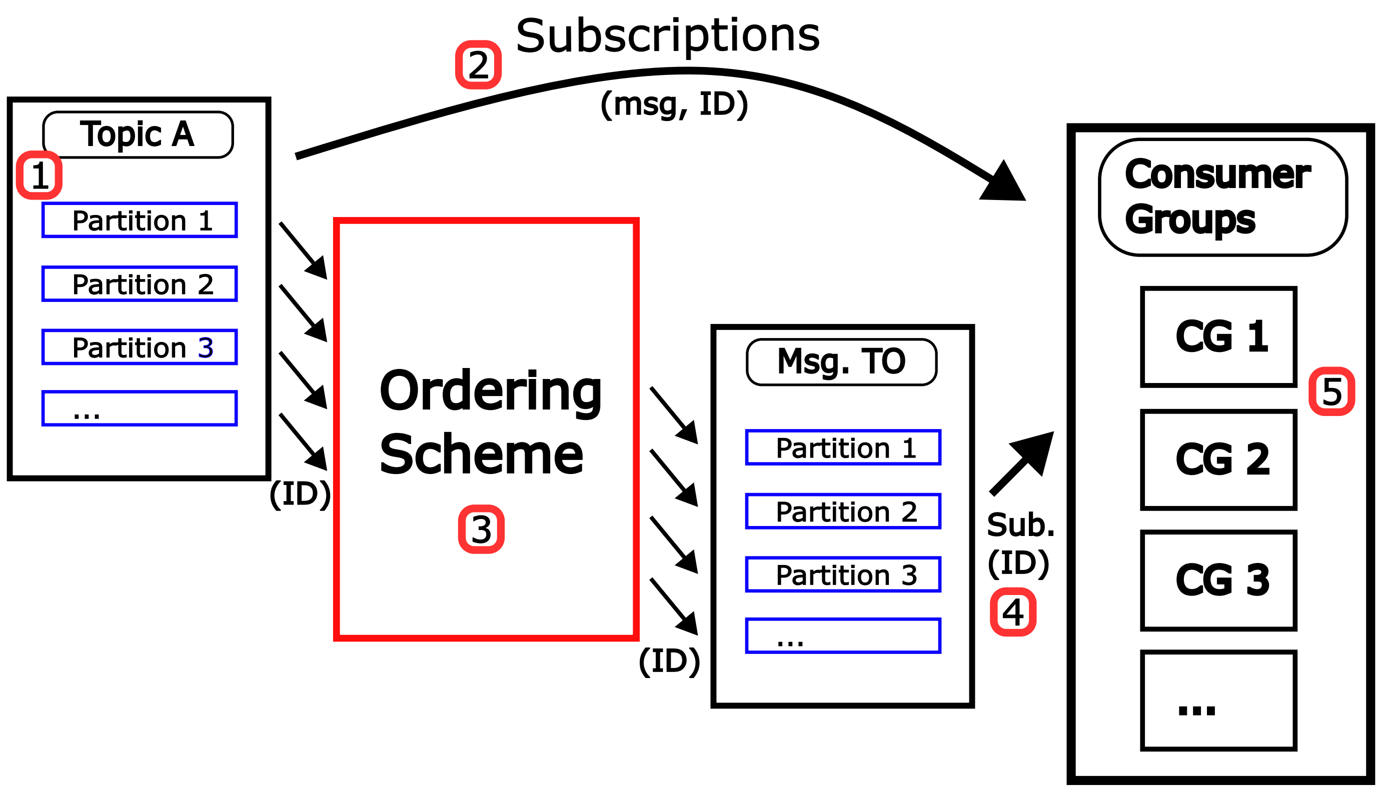
- Producers can create multiple data streams. Each data stream is input to a partition within a topic.
- Partitioning provides scalability. However, if multiple consumer groups are subscribed to the same subset of partitions, they do not consume the messages in the same order.
  ➜ Need of a new **ordering scheme**.

Model assumptions

- Apache Kafka replicates partitions, thus providing fault-tolerance.
- Channels are FIFO and reliable.

➤ Setup of data pipeline:

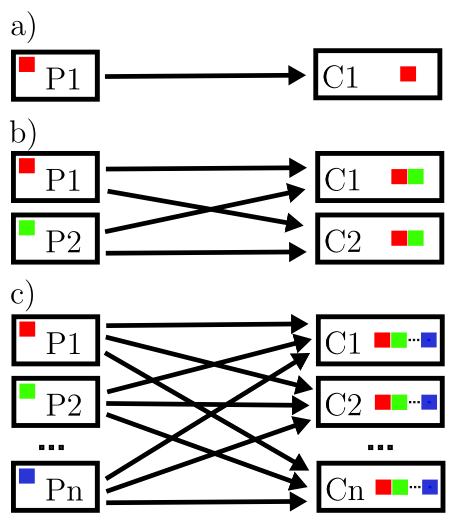
1. Message-ID tuples (msg, ID) are stored within the partitions of topic A.
2. Consumer groups are subscribed to topic A. Consumers within their group are assigned to partitions of topic A. ➜ Subscription link: Sends (msg, ID) tuples.
3. The hierarchical ordering scheme, orders the IDs and produces them to the topic "Msg. TO".
4. Consumer groups are also subscribed to "Msg. TO".
5. Consumers deliver a received message (msg, ID) as soon as they also received the ID via the ordering scheme.



Numerical results

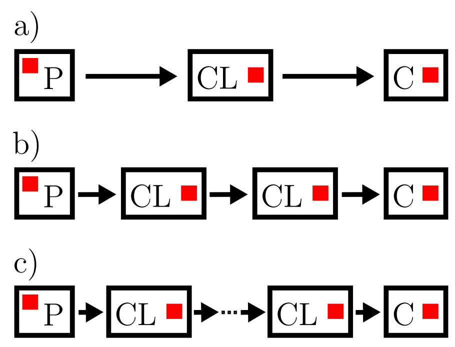
➤ Data sets are generated according to their number of messages `num_msg` $\in [10^2, 10^4]$ and message size `msg_size` $\in [10^4, 10^7]$ [B]. The ratio $r = \log(\texttt{num\_msg})/\log(\texttt{msg\_size})$ is characteristic for a data set and is used to label.

Kafka without TO



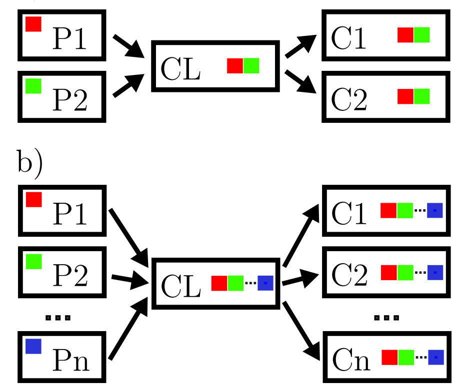
All consumers (C) are subscribed to all partitions (P). a) 1 P, 1 C; b) 2 P, 2 C; c) n P, n C, where $n \in \{4, 8, 16, 32\}$.

TO w. tree height

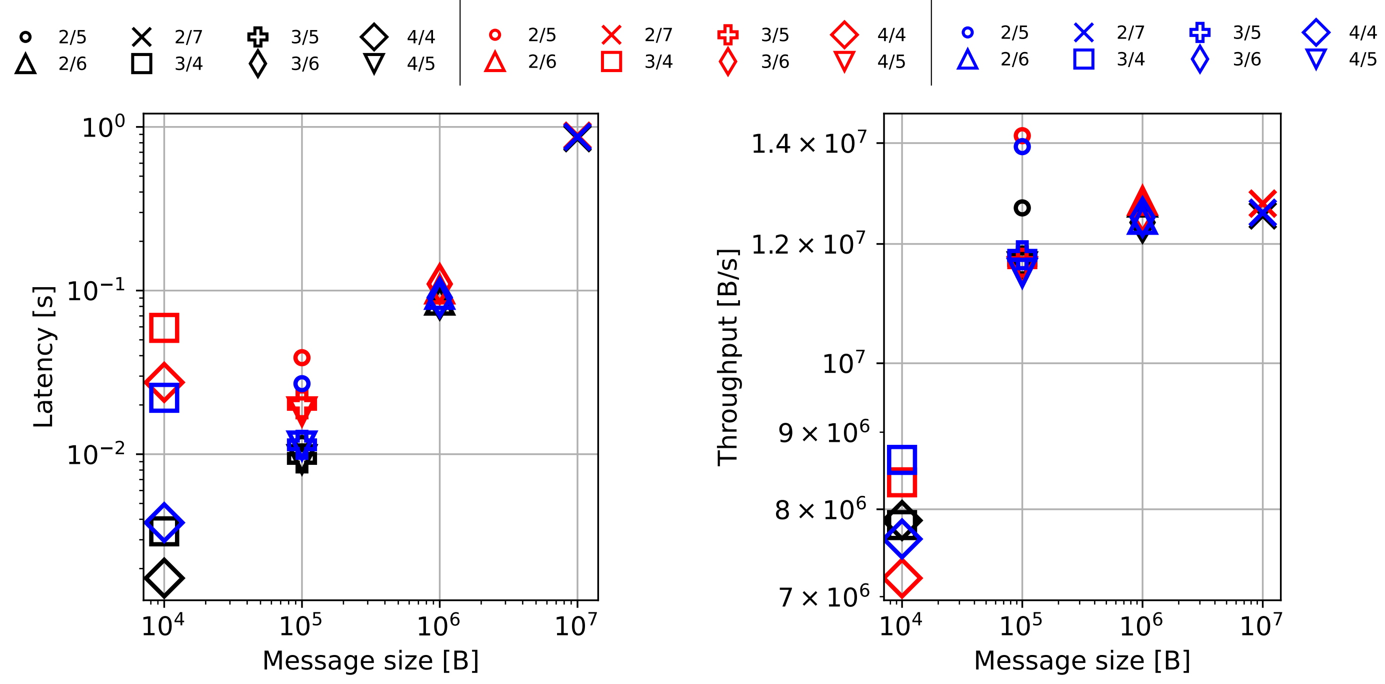


TO is ensured via the ordering scheme. The number of collectors in between is a) 1, b) 2, c) $n \in \{4, 8, 16, 32\}$.

TO w. bottleneck size



TO is ensured via the ordering scheme. The size of the bottleneck ranges from 1, via a) 2, to b) $n \in \{4, 8, 16, 32\}$.



➤ Each data point represents the mean of all configurations (either 1, 2, ..., n P & C or 1, 2, ..., n CL) of a topology (Kafka without TO, TO w. tree height, TO w. bottleneck size), since different configurations show no clear trend.

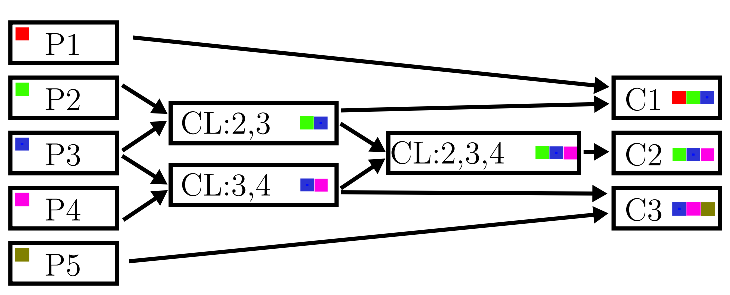
➤ For small `msg_size` the latency is increased by 5× to 10×, due to Kafka's batching for small and many messages. There is no latency increase for large `msg_size`.

➤ There is no throughput decrease when the Kafka is expanded with TO across partitions.

So far this scheme is not suitable for latency critical applications when $r \geq 10$. Kafka's intrinsic properties have to be further explored, especially its batching of messages.

Ordering Schemes & Implementation

Hierarchical scheme



Toy example: Consumer (C) C1 consumes partitions (P) P1, P2 and P3. C2 consumes P2, P3 and P4. C3 consumes P3, P4, P5. The arrangement of the tree is an optimization problem. [2]

➤ Consumers are subscribed to partitions. All their common interests have to follow a total order.

➤ Msg-IDs are send from partitions to consumers via layers formed by collectors (CL). The collectors resemble common interests.

➤ A consumer receives its order by having subscribed to the collectors that resemble its common interest ➜ ordering of the msg-IDs.

Merge algorithm

➤ If the lower level collectors share one or multiple partitions, the streams have to be merged, such that the lower level streams order is preserved.

Algorithm 1 Merge algorithm

Input: Input streams $s_i, i \in [1, ..., N]$, set of shared partitions S
Output: Merged stream preserving TO

$S_{count} \leftarrow 0$;
$s_{1,count} \leftarrow 0, ..., s_{N,count} \leftarrow 0$;
$q_{s_1} \leftarrow \varnothing, ..., q_{s_N} \leftarrow \varnothing$; ▷ queues for each s
for msg in consumer **do**
  Match `msg` to its q and s_{count}
  if $s_{count} == S_{count}$ and `msg.partition` $\notin S$ **then**
    send(msg)
  else
    q.append(msg)
    if `msg.partition` $\in S$ **then**
      $s_{count} \leftarrow s_{count} + 1$
  if $s_{i,count} > S_{count}\ \forall i \in [1, 2, ..., N]$ **then**
    $\forall$ q: *send*(msg) until `msg.partition` $\in S$
    send(msg), where `msg.partition` $\in S$ ▷ del. same `msg` in all other q
    $S_{count} \leftarrow S_{count} + 1$
    $\forall$ q: *send*(msg) until `msg.partition` $\in S$ or $q == \varnothing$

Implementation

- Python interface using `kafka-python`.
- Work in progress:
  - Move to distributed system (e.g. `Confluent Cloud`).
  - Implement scheme using `kafka streams`.

References

[1] The Apache Software Foundation. *https://kafka.apache.org*, 2024.

[2] Paulo Coelho, Tarcisio Ceolin Junior, Alysson Bessani, Fernando Dotti, and Fernando Pedone. Byzantine fault-tolerant atomic multicast. *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018.

Student researcher project by Ekkehard Steinmacher.
ekkehard.steinmacher@usi.ch